CS211, Lecture 21
Linear Sequence Structures

Announcements:

Overview:

- Motivation
- What's a sequence structure?
- Stack
- Queue
- Deque

1. Motivation

1.1 Sequence Structure

- want ADT to store and retrieve items
- what do you to create and manipulate a pile of things?
  - **`put`**: store current item
  - **`get`**: retrieve an item
- not interested in search
  - so, order not important
  - need to determine *where* **`put`** puts and **`get`** gets

1.2 Interface to implement

```
interface SeqStructure {

   // stick into sequence structure:
   void put(Object o);

   // extract from sequence structure
   Object get();

   boolean isEmpty();
   int size();
}
```

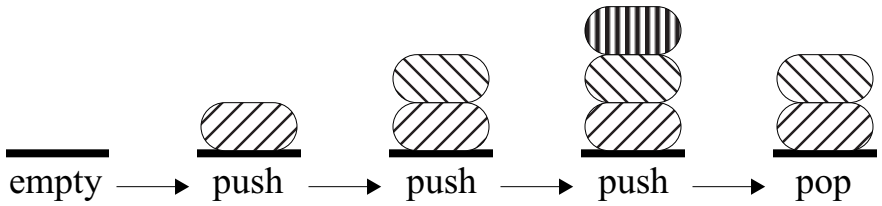
1.3 Linear Sequence Structures

- stack: LIFO
- queue: FIFO
- deque
- priority queue (we'll see a hierarchical version, too, so saving this for later)

2. Stack

2.1 Stack: LIFO

- Last in, first out:
  - push item on top of pile (put)
  - pop itemfrom top of pile (get)
- Visualize process:



2.2 Stack Pointer

- ***SP***: Address of *next* free element
- After pushing, **SP** increments (**SP++**):

2.3 Implementations

- Array: not dynamic, which means…
- List: better, since stack is dynamic

2.4 Stack size

- theoretically infinite
- realistically limited
- heap and stack space is finite for computer
- CS212: **`Runtime`** API and more…

3. Stack as Array

- Fields: array, SP, size
- Methods: see **`SeqStructure`**
- limitations: array size; could change to grow, though costly in terms of space (O(n))

3.1 Stubbed Stack

```
public class StackAsArray implements SeqStructure {

   private class SortedArray { }
   private SortedArray stack; // data in stack
   private int SP; // points to first empty cell
   public StackAsArray(int size) { }
   public void put(Object o) { }
   public Object get() { return null; }
   public int size() { return SP; }
   public boolean isEmpty() { return (SP == 0); }
   public String toString() { return ""; }

}
```

3.2 Member Class

- design: want to store elements in generic array
- could use a field that's an array
- I use an inner class to help keep track of size info

```
private class SortedArray {
   public Object[] a;
   public int MAXSIZE;
   public SortedArray(int n) {
      a = new Object[n];
      MAXSIZE = n;
   }
}
```

3.3 Constructor

```
public StackAsArray(int size) {
   stack = new SortedArray(size);
}
```

3.4 Fields

- Reference to sorted array (could scrap this and directly use an array!)
- **SP**: address in array of first free location
  - 0 is at bottom, which means empty stack
  - **MAXSIZE** is at top, which means full stack

```
private SortedArray stack; // data in stack
private int SP; // address of 1st free location
```

3.5 Put

- Algorithm:
  - check if stack is full (SP equals size)
  - if stack not full, add element at current SP and increment SP afterwards
- Code:

```
public void put(Object o) {
   if (SP == stack.MAXSIZE) {
      System.out.println("Stack overflow");
      return;
   }
   stack.a[SP] = o; // insert element
   SP++; // move SP "up"
}
```

- Alternatives?
  - return ***Something*`Exception`** to handle full case
  - syntax trick: **`stack.a[SP++]`**

3.6 Get

- Algorithm:
  - check if stack is empty
  - if not, extract object from top of stack, move SP down, and return the object
- Code:

```
public Object get() {
   if (SP == 0) {
      System.out.println("Empty stack!");
      return null;
   }
   Object temp = stack.a[--SP];
   stack.a[SP] = null;
   return temp;
}
```

3.7 Other operations

```
// size of stack is SP
public int size() {
   return SP;
}

// SP is at bottom:
public boolean isEmpty() {
   return (SP == 0);
}

// Stingify Stack:
public String toString() {
   String s = "LIFO: [";
   for (int i = 0; i < SP; i++) {
      s += stack.a[i];
         if (i < SP-1) s += ",";
   }
   s +="]";
   return s;
}
```

3.8 API `Stack`

- extends **`Vector`**
- has additional methods:
  - empty: is stack empty?
  - peek: inspect top of stack without popping
  - pop: take from top of stack
  - push: add to top of stack
  - search: “returns the 1-based position where an object is on this stack.”

4. Stack As List

4.1 Need to add/change some SLL operations:

```
// help to allow printing of empty stack:
public String toString() {
   if (head==null) return null;
   return head.toString();
}

// Adds element to head of list:
public boolean prepend(Object o) {
   ListNode tmp = new ListNode(o, head);
   if (head == null) tail=tmp;
   head = tmp;
   return true;
}
```

4.2 Implementations

- **`StackAsList`**: uses modified **`SLL`**
- **`TestStackAsListAlt`**: "bury" list into Stack class

4.3 Implementation

```
public class StackAsList implements SeqStructure {
   private SLL list;
   private int SP;
   public StackAsList() { list = new SLL(); }

   public void put(Object o) {
      list.prepend(o); SP++;
   }

   public Object get() {
      if (isEmpty()) {
         System.out.println("Empty list!");
         return null;
      }
      Object result = list.getHead().getItem();
      list.remove(result); // remove head of list
      SP--; // adjust SP
      return result;
   }

   public Object peek() {
      if (isEmpty()) return null;
      return list.getHead();
   }

   public boolean isEmpty() { return (SP==0); }
   public int size() { return SP; }

   public String toString() {
      return "LIFO: ["+list+"]";
   }
}
```

4.4 An Alternative Implementation

```
class StackAsListAlt implements SeqStructure{
   private class ListNode {
   } // not shown
   private ListNode list;
   public StackAsListAlt() { }

   public void put(Object o) {
      list = new ListNode(o,list); }

   public Object get() {
      if (! isEmpty()) {
         Object v = list.item;
         list = list.next;
         return v;
      } else {
         System.out.println("Empty!"); return null; }
      }

   public boolean isEmpty() { return (list == null); }

   public int size() {
      int v = 0;
      ListNode finger = list;
      while (finger != null) {
         v++; finger = finger.next;
      }
      return v;
   }

   public String toString() {
   } // not shown
}
```

4.5 Example Session

See **`TestStack.java`**:

```
   System.out.println("Testing list:");
   StackAsList s2 = new StackAsList();
   s2.put("Billy");
   s2.put("Rilly");
   s2.put("Silly");
   s2.put("Willy");
   System.out.println(s2);
   s2.get();System.out.println(s2);
   s2.get();System.out.println(s2);
   s2.get();System.out.println(s2);
   s2.get();System.out.println(s2);
   s2.get();

/* Output:
   Testing list:
   LIFO: [Willy Silly Rilly Billy]
   LIFO: [Silly Rilly Billy]
   LIFO: [Rilly Billy]
   LIFO: [Billy]
   LIFO: [null]
   Empty list!
*/
```

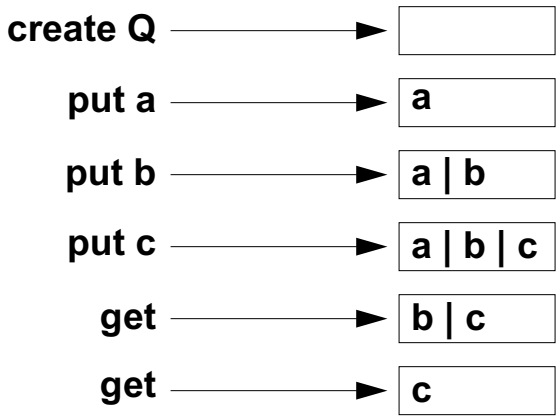
5. Queue

5.1 Singled-Ended Queue

- queue: “a line” (as in “to queue up”)
- what we’re calling queue
- “what goes in must come out…”
  - put/enqueue: add elements from one end
  - get/dequeue: remove elements from the other end

5.2 FIFO

- queue is FIFO
- ***FIFO***: first in, first out



5.3 Implementations

- array: **`QueueAsArray`**
- list: **`QueueAsList`**
- testing: **`TestQueue`**

6. Queue As Array

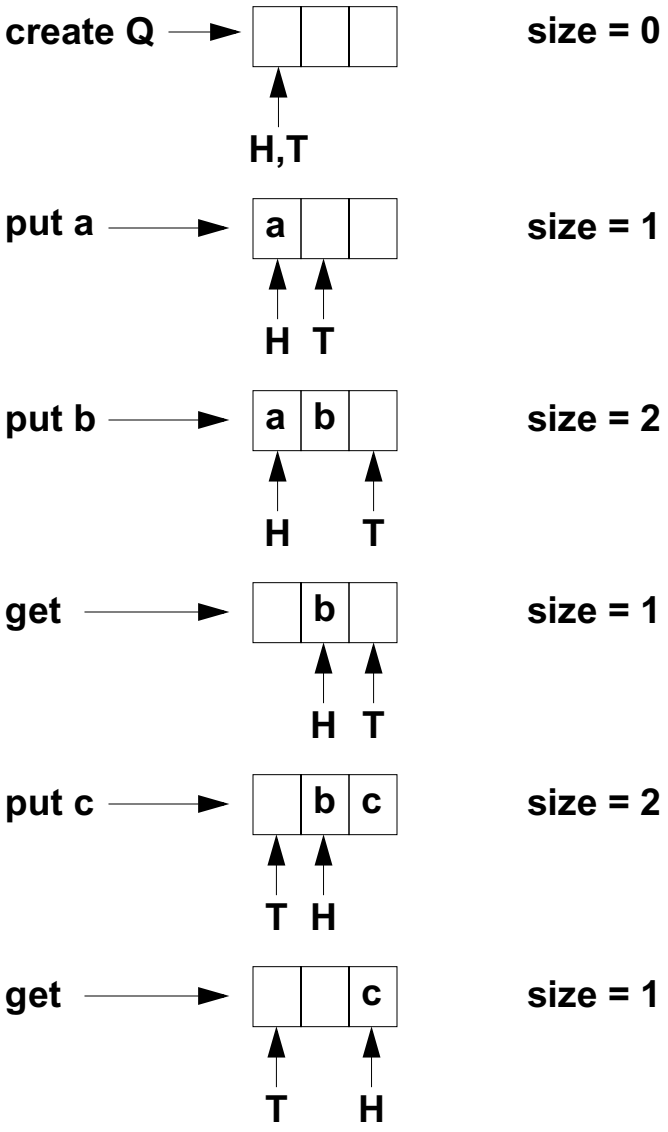
6.1 Design

- need to account for array filling…options:
  - allow only elements to fit from front to end
  - add space to array as needed
  - use circular array
- use circular array because of more flexibility and less wasted space
- what's a ***circular array***?

6.2 Circular Array and Queue

- keep track of head and tail of queue
  - head: first element to take
  - tail: first place to put an element
- In terms of FIFO:
  - get first element, which is stored at head
  - put element into first free position, which is tail
- tail and head move left to right and wrap around…

6.3 Circular Array Design



6.4 Fields

- **`Object a[]`**: store the elements
- **`head`**
- **`tail`**

6.5 Constructor

- create array of an input size (**`MAXSIZE`**)
- number of elements (**`size`**) must not exceed **`MAXSIZE`**
- ensure that **`head`** and **`tail`** are properly set, though they do not need to start at front of array

```
public class QueueAsArray implements SeqStructure {

   private int head; // points to element for deQ
   private int tail; // points to empty slot for enQ
   private Object[] a; // array of objects
   private int size;   // current # of objects in Q
   private final int MAXSIZE;  // max number of objects

   public QueueAsArray(int size) {
      a = new Object[size];
      MAXSIZE = size;
      head = 0;
      tail = 0;
      this.size = 0;
   }
   // methods: put, get, others
}
```

6.6 Put

- Algorithm:
  - prevent overfilling array (**`size`** > **`MAXSIZE`**)
  - insert item at **`tail`** location
  - increment **`tail`** to next element
  - if **`tail`** goes past end of queue, move to front (index 0)
  - increment count of items (**`size`**)
- Code:

```
public void put(Object o) {
   if (size == MAXSIZE) {
       System.out.print("Overflow!");
       return;
   }
   // queue isn't full,
   // so tail must point to an empty slot

   // insert item at end of Q and move tail up:
   a[tail++] = o;

   // if tail at end, move to front:
   if (tail == MAXSIZE) tail = 0;

   // increment count of items in Q:
   size++;
}
```

6.7 Get

- Algorithm:
  - prevent accessing empty array (**`size == 0`**)
  - get element from **`head`** (first element)
  - reset that location and increment head
  - if **`head`** goes past end of queue, move to front
  - decrease count of items (**`size`**)
- Code:

```
public Object get() {
   if (size == 0) {
       System.out.print("Empty!");
       return null;
   }
   // Since size is not 0,
   // we know head must point to a valid item

   // get item from head of Q (FIFO):
   Object temp = a[head];

   // reset that location and move head pointer:
   a[head++] = null;

   // wrap-around: if head at end move to front:
   if (head == MAXSIZE) head = 0;

   // decrease count of items in Q and return elem:
   size--;
   return temp;
}
```

6.8 Other Operations

```
// elements in queue:
public int size() {
   return size;
}

// is size==0?
public boolean isEmpty() {
   return (size == 0);
}

// Strigify queue:
public String toString() {
   String s = "FIFO: [";
   // there is at least one item in queue:
   if (size!=0) {
      int index = head;
      do {
         s += a[index++];
         if (index == MAXSIZE) index = 0;
         if (index != tail) s+=" | ";
      } while (index != tail);
   }
   s +="]";
   return s;
}
```

7. Queue As List

7.1 Use SLL

- no need to preallocate space because list is dynamic
- many operations already defined in **`SLL.java`**

7.2 Fields

- **`SLL list`**, which maintain most everything else
- **`size`**: help to keep track of Q

7.3 Operations

- **`put`**:
  - append to end of list, which automatically sets tail pointer
  - no need to check for maxsize
- **`get`**:
  - get head of list, then remove current head to move head pointer to next element
  - need to handle empty queue

7.4 Implementations

- **`QueueAsList`**
- **`TestQueueAsListAlt`**

7.5 Code

```
public class QueueAsList implements SeqStructure{

   private SLL list;
   private int size;

   public QueueAsList() { list = new SLL(); }

   public void put(Object o) {
      list.add(o);
      size++;
   }

   public Object get() {
      if (isEmpty()) {
         System.out.print("Empty!");
         return null;
      }
      Object o = list.getHead().getItem();
      list.remove(o);
      size--;
      return o;
   }

   public boolean isEmpty() { return list.isEmpty(); }

   public int size() { return size; }

   public String toString() {
      return "FIFO: ["+list+"]";
   }

}
```

7.6 Implementation

```
System.out.println("Test QueueAsList");
QueueAsList QAL = new QueueAsList();
System.out.println(QAL);
for (int i = 0; i <= 3; i++) {
    String item = "a"+i;
    System.out.print("Attempt to enqueue "+item+": ");
    QAL.put(item);
    System.out.println(" "+QAL);
}
System.out.println("Done putting!");
System.out.print("Get: "+QAL.get()+" ");
System.out.println(QAL);
System.out.print("Get: "+QAL.get()+" ");
System.out.println(QAL);
System.out.print("Get: "+QAL.get()+" ");
System.out.println(QAL);
System.out.print("Get: "+QAL.get()+" ");
System.out.println(QAL);
System.out.println("Done with lists!");

/* Output:
    Test QueueAsList
    FIFO: [null]
    Attempt to enqueue a0:  FIFO: [a0]
    Attempt to enqueue a1:  FIFO: [a0 a1]
    Attempt to enqueue a2:  FIFO: [a0 a1 a2]
    Attempt to enqueue a3:  FIFO: [a0 a1 a2 a3]
    Done putting!
    Get: a0 FIFO: [a1 a2 a3]
    Get: a1 FIFO: [a2 a3]
    Get: a2 FIFO: [a3]
    Get: a3 FIFO: [null]
    Done with lists!
*/
```

7.7 API Queue

- sorry, it's not there
- use **`LinkedList`** and your knowledge of FIFO

8. Deque

8.1 Double-Ended Queue (deque)

- remove items from both ends
- get and put for both head and tail
- see DS&A 6.3

9. Exercises

- Rewrite **`StackAsArray`** to issue a **`MyStackOverFlow`** exception if the memory is exceeded.
- Rewrite **`StackAsArray`** such that the array "grows" if the space is exhausted instead of issuing a complaint.
- Write a stack iterator for both the list and array implementations.
- Can you use head and tail in a queue to compute the size of the queue?
- Use mod (**`%`**) to assist with the head and tail updates in the **`QueueAsArray`** methods.
- Implement a queue with a circular list. Normally, this would be pointless, but it might make for good practice.